For Loops

15-110 – Monday 09/16

Learning Goals

Use **programming** to specify algorithms to computers

- Use **for loops** to repeat actions a specified number of times
- Use **for-each loops** to loop over **iterable** objects

For Loops

For Loops for Repeated Actions

We've learned how to use while loops and loop variables to iterate until a certain condition is met. When that condition is straightforward (increase a number until it reaches a certain limit), we can use a more standardized structure instead.

A **for-range loop** tells the program exactly how many times to repeat an action. The loop variable is updated by the loop itself!

```
for <loop_variable> in range(<max_num_plus_one>):
    <loop_body>
```

While Loops vs. For Loops

To sum the numbers from 0 to n in a while loop, we would write the following:

```
i = 0
result = 0
while i <= n:
    result = result + i
    i = i + 1
print(result)
```

In a for-range loop, we'll automatically start the loop variable at 0, and it will automatically increase by 1 each loop.

```
result = 0
for i in range(n+1):
    result = result + i
print(result)
```

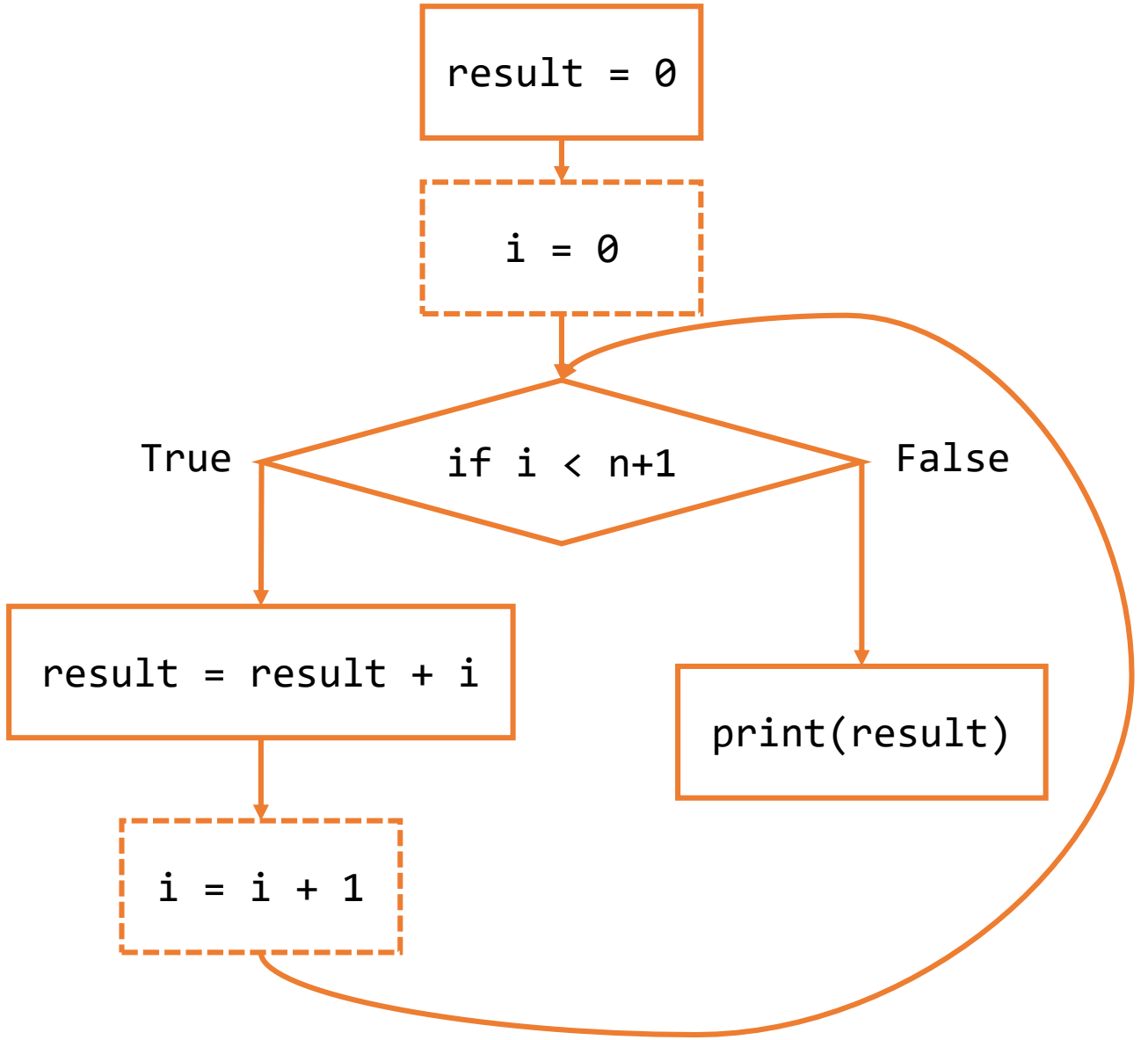
Note that we have to go up to n+1 because for-range goes **up to but not including** the given number. It's like saying while i < n+1.

For Loop Flow Chart

Unlike with while loops, we don't need to update the loop variable. The range we iterate over will do the update automatically.

We show actions done by the range with a dotted outline here, because they're **implicit**, not written directly.

```
result = 0
for i in range(n+1):
    result = result + i
print(result)
```



Range creates variable values

When we say `for i in range(10)`, `range(10)` generates all the values that the variable i will eventually hold. This is how i knows which value it should update to hold next each iteration.

We can update range to give it more information! If we call range on two numbers, it will start i at the first number and end i just before the last. So the following code would generate the numbers 3, 4, 5, 6, and 7.

```
for i in range(3, 8):
    print(i)
```

Range has a step

If we want to get really fancy, we can add a third argument to the range() function. This last argument is the **step** of the range, or how much the number should increase by each time. The following example would print the even numbers from 1 to 10:

```
for i in range(2, 11, 2):
    print(i)
```

Anything we can do in a for loop can also be done in a while loop. In a while loop, this would be equivalent to:

```
i = 2
while i < 11:
    print(i)
    i = i + 2
```

Example: Countdown

What if we want to write a program that counts backwards from 10 to 1? We can do that with range!

```
for i in range(10, 0, -1):
    print(i)
```

Note that i has to start at 10 and end at 0, in order to make 1 the last number that is printed.

Nesting Loops

Last week, we went over how we can use nesting to put conditionals in other conditionals or in while loops. We can put conditionals in for loops too, but more importantly, we can also nest loops inside of loops!

We mostly do this with for-range loops, and mostly when we want to loop over *multiple dimensions*.

```
for <loop_var_1> in range(<end_num_1>):
    for <loop_var_2> in range(<end_num_2>):
        <both_loops_body>
    <just_outer_loop_body>
```

When we nest loops, we repeat the inner loop **every time** the outer loop takes a step.

Nested Loops Example: Coordinates

For example, let's say we want to print all the coordinates on a plane from (0,0) to (5,5)

```
for x in range(5):
    for y in range(5):
        print("(", x, ",", y, ")")
```

Note that every iteration of y happens anew in each iteration of x.

Nested Loops Example: Multiplication Table

The following code prints out a 4x4 multiplication table. We can use **code tracing** to find the values at each iteration of the loops.

```
for x in range(1, 5):
    for y in range(1, 5):
        print(x,"*",y,"=",x*y)
```

Iteration	x	y	result
1	1	1	1
2	1	2	2
3	1	3	3
4	1	4	4
5	2	1	2
6	2	2	4
7	2	3	6
8	2	4	8
9	3	1	3
10	3	2	6
11	3	3	9
12	3	4	12
13	4	1	4
14	4	2	8
15	4	3	12
16	4	4	16

Activity: Kahoot Quiz

Predict what the loop will print based on its range!

Iterable Values

There are some types of values that can be thought of as a whole composed of many parts. We call these types **iterable**, because we can iterate over each of the parts.

Of the types we've explored so far, **strings** are iterable, because they are composed of **characters** (individual letters or symbols).

For Loops with Iterable Values

We can also use for loops to iterate over these values that are iterable! Using a for loop, we can write a program that loops over each of the characters of a string in order.

```
for <character_variable> in <string>:
    <character_action_body>
```

For example, if we run the following code, it will print out each character of the string with an exclamation point after it.

```
for c in "Hello":
    print(c + "!")
```

This leads us to new problems we can solve by breaking up strings into characters...

Linear Search

Searching for Characters

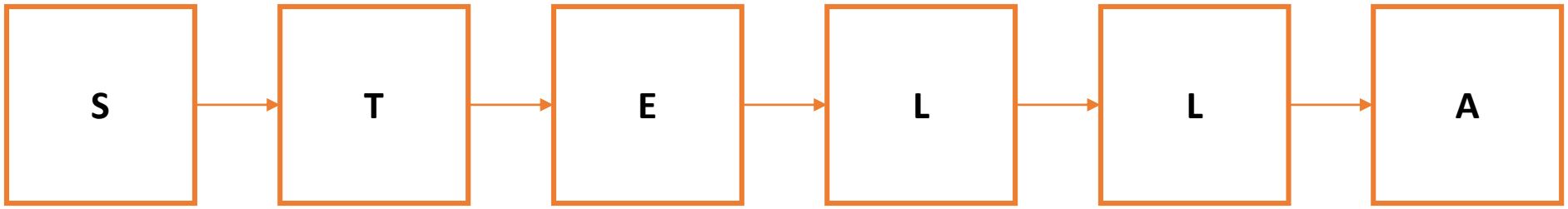
Say we want to determine whether a string contains a specific character. How can we do that?

We'll need to view this from a computer's perspective...

How Computers See Characters

If we ask a computer to see if a character is in a string, it sees the whole strings as a series of not-yet-known characters:

In order to determine if the character is one of them, it needs to check each character in turn!



Search Function

We can use a for-each loop to implement this approach as code!

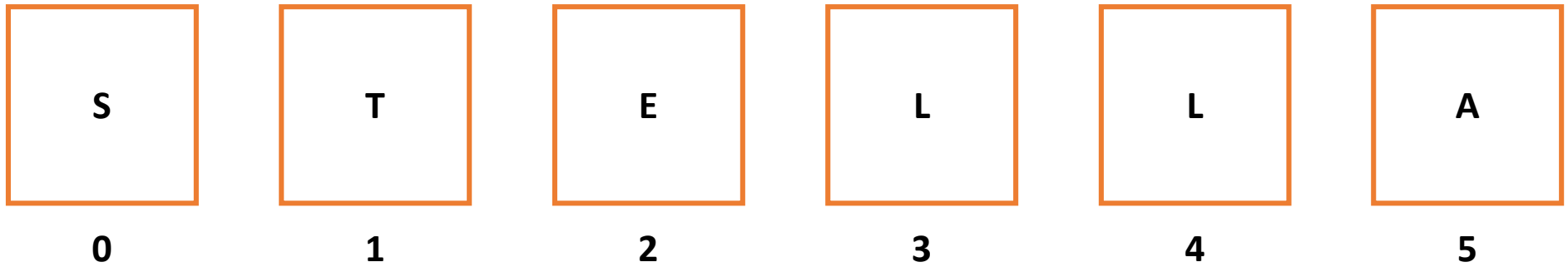
```
def containsChar(s, c):
    for letter in s:
        if letter == c:
            return True
    return False
```

Note that we can return True as soon as we find the character, but we can't return False until we've gone through all the characters.

Finding a Location

Now we can find if a character is in a string, but we don't know where it is. How can we find the character's position?

First, we need to determine what each character's position is. We can assign integer positions in order, starting with 0.



Getting Characters By Location

If we know a character's position, Python will let us access that character directly from the string! We need to use **square brackets** with the position in between to get the character.

```
s = "STELLA"
c = s[2] # "E"
```

Recall that we can get the number of characters in a string with len(s). This will come in handy soon.

Search Function – Take Two

Now we can write the search function a second time. This time, we'll write it as a **for-range loop**, where the index of the loop is each position in the string!

Instead of returning True/False, we'll return the index. If we don't find the character, we'll return -1.

```
def charLocation(s, c):
    for i in range(len(s)):
        if s[i] == c:
            return i
    return -1
```

Search Function – Take 3 and Take 4

Anything that can be done as a for loop can also be done with a while loop!

```
def charLocation(s, c):
    i = 0
    while i < len(s):
        if s[i] == c:
            return i
        i = i + 1
    return -1
```

We could also have just added a counter variable to the for-each loop.

```
def charLocation(s, c):
    i = 0
    for letter in s:
        if letter == c:
            return i
        i = i + 1
    return -1
```

Which of the four approaches that we've gone over is best? It depends!

Learning Goals

Use **programming** to specify algorithms to computers

- Use **for loops** to repeat actions a specified number of times
- Use **for-each loops** to loop over **iterable** objects